**Image Captioning using the Flickr8k Dataset**

**Context**

A new benchmark collection for sentence-based image description and search, consisting of 8,000 images that are each paired with five different captions which provide clear descriptions of the salient entities and events. … The images were chosen from six different Flickr groups, and tend not to contain any well-known people or locations, but were manually selected to depict a variety of scenes and situations

**1. Function: show\_image**

This function takes in an image tensor (img) and optionally a title to display the image. Here's the breakdown:

def show\_image(img, title=None):

"""Imshow for Tensor."""

* **Purpose:** The function displays a tensor image using plt.imshow().
* **Parameters:**
  + img: The image tensor to be shown.
  + title: Optional title for the image (default is None).

**Inside the function:**

* **Unnormalize the image:**

img[0] = img[0] \* 0.229

img[1] = img[1] \* 0.224

img[2] = img[2] \* 0.225

img[0] += 0.485

img[1] += 0.456

img[2] += 0.406

* + When working with image data in deep learning, images are often normalized before being fed into models. This part "unnormalizes" the image so that it can be displayed properly.
  + The normalization values 0.229, 0.224, 0.225 and offsets 0.485, 0.456, 0.406 are standard values used in ImageNet-pretrained models (e.g., ResNet).
  + It scales the RGB channels back to their original range (before normalization).
* **Convert tensor to a NumPy array and transpose dimensions:**

img = img.numpy().transpose((1, 2, 0))

* + Converts the PyTorch tensor into a NumPy array.
  + The .transpose((1, 2, 0)) rearranges the image's dimensions from (C, H, W) (Channel, Height, Width) to (H, W, C), which is the format expected by matplotlib for displaying images.
* **Display the image using Matplotlib:**

plt.imshow(img)

if title is not None:

plt.title(title)

plt.pause(0.001)

* + plt.imshow(img) displays the image.
  + plt.title(title) sets the title if provided.
  + plt.pause(0.001) allows for small delays to ensure the plot gets updated.

**2. Transform Pipeline:**

The next section defines the transforms to be applied to the images during preprocessing:

transforms = T.Compose([

T.Resize(226),

T.RandomCrop(224),

T.RandomHorizontalFlip(),

T.ToTensor(),

T.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))

])

* **T.Resize(226)**: Resizes the image to 226 pixels on the shorter side.
* **T.RandomCrop(224)**: Crops a random 224x224 pixel region from the resized image.
* **T.RandomHorizontalFlip()**: Randomly flips the image horizontally with a 50% chance, useful for data augmentation.
* **T.ToTensor()**: Converts the image to a PyTorch tensor and scales pixel values from [0, 255] to [0, 1].
* **T.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))**: Normalizes the image using mean and standard deviation values. These are typical values for models pre-trained on ImageNet.

**3. Creating Dataset:**

A custom dataset class (FlickrDataset) is instantiated using the following:

dataset = FlickrDataset(

root\_dir = data\_location + "/Images",

caption\_file = data\_location + "/captions.txt",

transform = transforms

)

* **root\_dir**: Path to the image directory.
* **caption\_file**: Path to the captions file.
* **transform**: The transformation pipeline (defined above) to be applied to each image.

This dataset is likely responsible for loading the Flickr images and their associated captions, applying the transforms as necessary.

**4. Dataloader Setup:**

A PyTorch DataLoader is created to efficiently load batches of data:

data\_loader = get\_data\_loader(

dataset = dataset,

batch\_size = BATCH\_SIZE,

num\_workers = NUM\_WORKER,

shuffle = True

)

* **dataset**: The dataset object defined earlier.
* **batch\_size**: Number of samples per batch, controlled by the BATCH\_SIZE variable.
* **num\_workers**: Number of subprocesses to use for data loading, controlled by NUM\_WORKER.
* **shuffle=True**: Randomly shuffle the dataset at every epoch to prevent model overfitting.

**5. Device Selection:**

The code checks if a GPU is available and sets the computation device accordingly:

device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

* **cuda:0**: Indicates the first GPU (if available).
* If a GPU is not available, it falls back to the CPU.

**6. EncoderCNN Class (Convolutional Neural Network Encoder)**

The EncoderCNN class is designed to extract image features using a **ResNet-50** architecture, which is a popular deep convolutional neural network pretrained on ImageNet. Here's what each part of the class does:

class EncoderCNN(nn.Module):

def \_\_init\_\_(self):

super(EncoderCNN, self).\_\_init\_\_()

resnet = models.resnet50(pretrained=True)

for param in resnet.parameters():

param.requires\_grad\_(False)

modules = list(resnet.children())[:-2]

self.resnet = nn.Sequential(\*modules)

**Key Steps:**

1. **Inheritance from nn.Module:** The class inherits from torch.nn.Module, making it a PyTorch neural network model.
2. **Load Pretrained ResNet-50 Model:**
   * The ResNet-50 model is loaded with pre-trained weights (pretrained=True).
   * This model has been trained on the ImageNet dataset and is used for feature extraction.
3. **Freezing the ResNet-50 Weights:**
   * The loop for param in resnet.parameters(): param.requires\_grad\_(False) disables gradient computation for the ResNet parameters. This means the ResNet weights will not be updated during training (which is common in transfer learning).
4. **Removing the Fully Connected Layers:**
   * modules = list(resnet.children())[:-2] extracts all layers of ResNet **except** for the last two layers (global average pooling and fully connected layers).
   * The purpose is to retain the convolutional layers that output feature maps, but exclude layers designed for classification.
5. **Sequential Model:**
   * self.resnet = nn.Sequential(\*modules) wraps the selected layers into a sequential model, which allows them to be treated as a single layer.

**Forward Pass:**

def forward(self, images):

features = self.resnet(images) #(batch\_size, 2048, 7, 7)

features = features.permute(0, 2, 3, 1) #(batch\_size, 7, 7, 2048)

features = features.view(features.size(0), -1, features.size(-1)) #(batch\_size, 49, 2048)

return features

1. **Forward Method:** The forward function defines how the model processes input during the forward pass.
2. **Extract Features:**
   * features = self.resnet(images) passes the input image through the ResNet layers, outputting a feature map of size (batch\_size, 2048, 7, 7), where 2048 is the number of channels, and 7x7 is the spatial resolution.
3. **Permute Dimensions:**
   * features.permute(0, 2, 3, 1) changes the dimensions to (batch\_size, 7, 7, 2048), swapping the channel dimension to the last axis, which makes it easier to work with.
4. **Flatten Features:**
   * features.view(features.size(0), -1, features.size(-1)) reshapes the feature map to (batch\_size, 49, 2048), where 49 comes from flattening the 7x7 spatial dimensions (7x7 = 49), and 2048 is the number of channels.
   * This produces a feature vector for each spatial position, which can be passed to the attention mechanism or further layers.

**7. Bahdanau Attention Class**

This class implements **Bahdanau attention**, which allows the model to focus on different parts of the input sequence (in this case, the image features) when generating the output.

class Attention(nn.Module):

def \_\_init\_\_(self, encoder\_dim, decoder\_dim, attention\_dim):

super(Attention, self).\_\_init\_\_()

self.attention\_dim = attention\_dim

self.W = nn.Linear(decoder\_dim, attention\_dim)

self.U = nn.Linear(encoder\_dim, attention\_dim)

self.A = nn.Linear(attention\_dim, 1)

**Key Components:**

1. **Inputs:**
   * encoder\_dim: The dimension of the encoder output (in this case, 2048 from the ResNet features).
   * decoder\_dim: The dimension of the hidden state from the decoder (e.g., an LSTM or GRU).
   * attention\_dim: The dimensionality of the attention mechanism, which is typically a smaller intermediate size.
2. **Linear Layers:**
   * self.W = nn.Linear(decoder\_dim, attention\_dim): This linear layer transforms the decoder hidden state into the attention space.
   * self.U = nn.Linear(encoder\_dim, attention\_dim): This linear layer transforms the encoder features into the attention space.
   * self.A = nn.Linear(attention\_dim, 1): This is the final layer that produces attention scores.

**Forward Pass:**

def forward(self, features, hidden\_state):

u\_hs = self.U(features) #(batch\_size, num\_layers, attention\_dim)

w\_ah = self.W(hidden\_state) #(batch\_size, attention\_dim)

combined\_states = torch.tanh(u\_hs + w\_ah.unsqueeze(1)) #(batch\_size, num\_layers, attention\_dim)

attention\_scores = self.A(combined\_states) #(batch\_size, num\_layers, 1)

attention\_scores = attention\_scores.squeeze(2) #(batch\_size, num\_layers)

alpha = F.softmax(attention\_scores, dim=1) #(batch\_size, num\_layers)

attention\_weights = features \* alpha.unsqueeze(2) #(batch\_size, num\_layers, features\_dim)

attention\_weights = attention\_weights.sum(dim=1) #(batch\_size, features\_dim)

return alpha, attention\_weights

1. **Encoder Features and Decoder Hidden State:**
   * The features represent the output from the encoder (ResNet), and the hidden\_state is the current hidden state from the decoder (such as an LSTM).
2. **Transform Features and Hidden State:**
   * u\_hs = self.U(features) applies a linear transformation to the encoder features (size: (batch\_size, num\_layers, attention\_dim)).
   * w\_ah = self.W(hidden\_state) applies a linear transformation to the hidden state (size: (batch\_size, attention\_dim)).
3. **Combine the States:**
   * combined\_states = torch.tanh(u\_hs + w\_ah.unsqueeze(1)) applies a non-linear transformation (tanh) after summing the transformed encoder features and decoder hidden state. The unsqueeze(1) ensures that the hidden state can be broadcasted across the time steps (or "layers").
4. **Calculate Attention Scores:**
   * attention\_scores = self.A(combined\_states) produces attention scores for each time step (or spatial position) in the encoder features. The size is (batch\_size, num\_layers, 1).
   * attention\_scores.squeeze(2) removes the last dimension, leaving (batch\_size, num\_layers).
5. **Compute Attention Weights:**
   * alpha = F.softmax(attention\_scores, dim=1) applies a softmax to normalize the attention scores into a probability distribution (alpha), where each element represents the attention weight for a specific time step/spatial position.
6. **Apply Attention Weights to Features:**
   * attention\_weights = features \* alpha.unsqueeze(2) multiplies the encoder features by their corresponding attention weights. The unsqueeze(2) ensures the attention weights are broadcasted correctly.
   * attention\_weights.sum(dim=1) sums over the time steps to produce the final context vector, which will be passed to the decoder.

**Output:**

* alpha: The attention weights (probability distribution over the encoder features).
* attention\_weights: The weighted sum of the encoder features, which serves as the context vector.

**8. Class Initialization:**

The constructor (\_\_init\_\_) initializes the various layers and parameters of the model.

class DecoderRNN(nn.Module):

def \_\_init\_\_(self, embed\_size, vocab\_size, attention\_dim, encoder\_dim, decoder\_dim, drop\_prob=0.3):

* **Parameters:**
  + embed\_size: The size of the word embeddings.
  + vocab\_size: The size of the vocabulary (number of unique words).
  + attention\_dim: The dimension of the attention mechanism.
  + encoder\_dim: The dimension of the encoder's output features (e.g., 2048 from the ResNet encoder).
  + decoder\_dim: The size of the LSTM's hidden state.
  + drop\_prob: Dropout probability for regularization (to prevent overfitting).

**Layers Initialized:**

self.embedding = nn.Embedding(vocab\_size, embed\_size)

self.attention = Attention(encoder\_dim, decoder\_dim, attention\_dim)

* **self.embedding:** This layer converts input words (captions) into dense word embeddings.
* **self.attention:** This is an instance of the attention mechanism (defined previously) that computes attention over image features.

self.init\_h = nn.Linear(encoder\_dim, decoder\_dim)

self.init\_c = nn.Linear(encoder\_dim, decoder\_dim)

self.lstm\_cell = nn.LSTMCell(embed\_size + encoder\_dim, decoder\_dim, bias=True)

self.f\_beta = nn.Linear(decoder\_dim, encoder\_dim)

* **self.init\_h:** Linear layer to initialize the LSTM hidden state h from the encoder's features.
* **self.init\_c:** Linear layer to initialize the LSTM cell state c from the encoder's features.
* **self.lstm\_cell:** LSTM cell that takes the word embeddings and attention context as input to generate the hidden states.
  + The input size is embed\_size + encoder\_dim because it concatenates the word embedding and context vector.
* **self.f\_beta:** Linear layer used to compute a "gate" for the attention weights (not used in the code you've provided but could be used to control how much attention is applied).

self.fcn = nn.Linear(decoder\_dim, vocab\_size)

self.drop = nn.Dropout(drop\_prob)

* **self.fcn:** Fully connected layer that transforms the LSTM hidden state into vocabulary-sized output (logits for each word).
* **self.drop:** Dropout layer for regularization.

**9. Forward Method:**

This function defines the forward pass for the decoder. It takes as input the image features (from the encoder) and the target captions (during training) to predict the next word in the sequence.

def forward(self, features, captions):

**Steps:**

embeds = self.embedding(captions)

* **Embeddings:** The captions are passed through the embedding layer to convert each word into a dense vector of size embed\_size.

h, c = self.init\_hidden\_state(features)

* **Initialize LSTM State:** The LSTM's hidden state (h) and cell state (c) are initialized using the image features (encoded by the encoder).

seq\_length = len(captions[0]) - 1

batch\_size = captions.size(0)

num\_features = features.size(1)

* **Sequence Length and Batch Size:** These are used to iterate over each word in the caption.

preds = torch.zeros(batch\_size, seq\_length, self.vocab\_size).to(device)

alphas = torch.zeros(batch\_size, seq\_length, num\_features).to(device)

* **Initialize Storage for Predictions and Attention Weights:**
  + preds: This will store the predicted word probabilities for each time step.
  + alphas: This will store the attention weights for each time step.

**Iterating Through Each Word in the Sequence:**

for s in range(seq\_length):

alpha, context = self.attention(features, h)

lstm\_input = torch.cat((embeds[:, s], context), dim=1)

h, c = self.lstm\_cell(lstm\_input, (h, c))

output = self.fcn(self.drop(h))

preds[:, s] = output

alphas[:, s] = alpha

* **Attention Mechanism:**
  + alpha, context = self.attention(features, h): The attention mechanism computes the attention weights alpha and the context vector context based on the encoder features (features) and the LSTM hidden state (h).
* **Concatenate Embedding and Context:**
  + lstm\_input = torch.cat((embeds[:, s], context), dim=1) concatenates the embedding of the current word (embeds[:, s]) and the attention context vector (context) to form the input to the LSTM.
* **LSTM Cell Update:**
  + h, c = self.lstm\_cell(lstm\_input, (h, c)): The LSTM cell is updated using the concatenated input, producing a new hidden state (h) and cell state (c).
* **Prediction Generation:**
  + output = self.fcn(self.drop(h)): The hidden state h is passed through a dropout layer and then through the fully connected layer (fcn) to predict the next word's probabilities over the vocabulary.
  + These predictions (preds[:, s]) and attention weights (alphas[:, s]) are stored at each time step.

return preds, alphas

* **Return Predictions and Attention Weights:** The model returns the predicted word probabilities (preds) and attention weights (alphas) for the entire sequence.

**10. Caption Generation (Inference):**

This method (generate\_caption) is used during inference to generate captions for an image based on the features extracted from the encoder.

def generate\_caption(self, features, max\_len=20, vocab=None):

* **Parameters:**
  + features: The image features from the encoder.
  + max\_len: The maximum length of the generated caption.
  + vocab: The vocabulary object to convert between words and indices.

**Steps:**

h, c = self.init\_hidden\_state(features)

word = torch.tensor(vocab.stoi['<SOS>']).view(1, -1).to(device)

embeds = self.embedding(word)

captions = []

* **Initialize LSTM State:** The hidden state (h) and cell state (c) are initialized using the image features.
* **Start Word (<SOS>):** The decoder starts generating the caption with the "Start of Sentence" token (<SOS>), which is embedded into a word vector.
* **Captions List:** An empty list is initialized to store the generated caption.

**Iterating to Generate Words:**

for i in range(max\_len):

alpha, context = self.attention(features, h)

lstm\_input = torch.cat((embeds[:, 0], context), dim=1)

h, c = self.lstm\_cell(lstm\_input, (h, c))

output = self.fcn(self.drop(h))

output = output.view(batch\_size, -1)

predicted\_word\_idx = output.argmax(dim=1)

captions.append(predicted\_word\_idx.item())

if vocab.itos[predicted\_word\_idx.item()] == "<EOS>":

break

embeds = self.embedding(predicted\_word\_idx.unsqueeze(0))

* **Attention Mechanism:** At each time step, attention is applied to the image features to compute a context vector.
* **LSTM Cell Update:** The LSTM hidden state is updated based on the previous word embedding and the context vector.
* **Generate Next Word:** The output of the LSTM is passed through the fully connected layer to produce a probability distribution over the vocabulary. The word with the highest probability is selected as the next word (predicted\_word\_idx).
* **Early Stopping with <EOS>:** If the model generates the <EOS> token, the loop stops early.
* **Update Embedding:** The predicted word is then embedded and used as the input for the next time step.

**Return the Generated Caption:**

return [vocab.itos[idx] for idx in captions], alphas

* The indices of the generated words are converted back to actual words using vocab.itos and returned along with the attention weights.

### 11. Initializing Hidden States:

This helper method initializes the hidden and cell states of the LSTM.

def init\_hidden\_state(self, encoder\_out):

mean\_encoder\_out = encoder\_out.mean(dim=1)

h = self.init\_h(mean\_encoder\_out)

c = self.init\_c(mean\_encoder\_out)

return h, c

* **Averaging Encoder Features:** The image features from the encoder are averaged across the spatial dimensions (mean\_encoder\_out = encoder\_out.mean(dim=1)), which helps initialize the hidden and cell states.
* **Linear Layers:** The averaged encoder output is passed through two separate linear layers (self.init\_h and self.init\_c) to generate the initial hidden state (h) and cell state (c) for the LSTM.

**12. EncoderDecoder Class:**

The EncoderDecoder class brings together the CNN-based encoder (EncoderCNN) and the RNN-based decoder (DecoderRNN) to form a complete image captioning model.

class EncoderDecoder(nn.Module):

def \_\_init\_\_(self, embed\_size, vocab\_size, attention\_dim, encoder\_dim, decoder\_dim, drop\_prob=0.3):

super().\_\_init\_\_()

self.encoder = EncoderCNN()

self.decoder = DecoderRNN(

embed\_size=embed\_size,

vocab\_size=len(dataset.vocab),

attention\_dim=attention\_dim,

encoder\_dim=encoder\_dim,

decoder\_dim=decoder\_dim

)

* **Parameters**:
  + embed\_size: The size of the word embeddings.
  + vocab\_size: The size of the vocabulary (number of unique words).
  + attention\_dim: The size of the attention mechanism.
  + encoder\_dim: The size of the encoder's output (e.g., the number of features output by ResNet, typically 2048).
  + decoder\_dim: The size of the LSTM's hidden state.
* **self.encoder**: An instance of EncoderCNN (which is defined separately) that extracts features from the input image. Typically, this would be a pretrained ResNet with the final fully connected layers removed.
* **self.decoder**: An instance of DecoderRNN, which generates a sequence of words (captions) for the given image features. This uses an LSTM with attention.

**Forward Pass:**

def forward(self, images, captions):

features = self.encoder(images)

outputs = self.decoder(features, captions)

return outputs

* **images**: The input images, which are passed through the encoder to extract **features**.
* **captions**: The target captions (in training mode) passed to the decoder, which generates the caption predictions for each word in the sequence.
* **features = self.encoder(images)**: The encoder processes the image to generate image features.
* **outputs = self.decoder(features, captions)**: The decoder uses the image features and captions to predict the next word at each time step. The output is a set of word predictions.
* **Returns**: The model returns the predicted word probabilities for the captions.

**13. Model Initialization and Hyperparameters:**

#Hyperparams

embed\_size = 300

vocab\_size = len(dataset.vocab)

attention\_dim = 256

encoder\_dim = 2048

decoder\_dim = 512

learning\_rate = 3e-4

* **Hyperparameters**:
  + embed\_size=300: Size of the word embeddings (each word is represented as a vector of 300 dimensions).
  + vocab\_size = len(dataset.vocab): Size of the vocabulary, determined by the dataset's vocabulary (number of unique words).
  + attention\_dim=256: Dimension of the attention layer, controlling the size of the attention mechanism.
  + encoder\_dim=2048: The size of the image features from the encoder (for ResNet50, this is typically 2048).
  + decoder\_dim=512: The size of the hidden state in the LSTM.
  + learning\_rate=3e-4: The learning rate for the optimizer (Adam in this case).

**14. Model Initialization:**

#init model

model = EncoderDecoder(

embed\_size=300,

vocab\_size=len(dataset.vocab),

attention\_dim=256,

encoder\_dim=2048,

decoder\_dim=512

).to(device)

* **Model Initialization**: An instance of the EncoderDecoder model is created using the defined hyperparameters and moved to the appropriate device (GPU or CPU).

**15. Loss Function and Optimizer:**

criterion = nn.CrossEntropyLoss(ignore\_index=dataset.vocab.stoi["<PAD>"])

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

* **Loss Function**:
  + nn.CrossEntropyLoss(ignore\_index=dataset.vocab.stoi["<PAD>"]): The loss function used is cross-entropy loss, which is common for classification tasks (here, predicting the next word in the caption).
  + **ignore\_index=dataset.vocab.stoi["<PAD>"]**: The loss function ignores the <PAD> token (which is used to pad captions to a fixed length) to prevent it from affecting the training.
* **Optimizer**:
  + optim.Adam(model.parameters(), lr=learning\_rate): The Adam optimizer is used to update the model's parameters based on the loss. The learning rate is set to 3e-4.

**16. Saving the Model:**

#helper function to save the model

def save\_model(model, num\_epochs):

model\_state = {

'num\_epochs': num\_epochs,

'embed\_size': embed\_size,

'vocab\_size': len(dataset.vocab),

'attention\_dim': attention\_dim,

'encoder\_dim': encoder\_dim,

'decoder\_dim': decoder\_dim,

'state\_dict': model.state\_dict()

}

torch.save(model\_state, 'attention\_model\_state.pth')

* **save\_model Function**: This helper function saves the model's state so that it can be reloaded later for inference or continued training. It saves:
  + num\_epochs: The number of epochs completed.
  + Hyperparameters (embed\_size, vocab\_size, attention\_dim, encoder\_dim, decoder\_dim).
  + model.state\_dict(): The learned parameters of the model.
* The saved model state is stored in a file named attention\_model\_state.pth.

**16. Loading the Model:**

* def load\_model(model): The function takes model as an input, which is a neural network model (likely a PyTorch model).

Checking if a Saved Model Exists:

* if os.path.exists('attention\_model\_state.pth'): This line checks if the file 'attention\_model\_state.pth' (which contains the saved model's state) exists in the current directory.
  + **.pth files** are commonly used in PyTorch to save model parameters.

If the Saved Model Exists:

* If the file exists, the function:
  + Prints "Loading saved model..." to inform the user.
  + Loads the checkpoint using torch.load('attention\_model\_state.pth'). This checkpoint typically contains:
    - The model's saved parameters (state\_dict).
    - Other metadata like the number of epochs (num\_epochs).
  + The line model.load\_state\_dict(checkpoint['state\_dict']) loads the model's saved parameters (weights and biases).
  + Then, it prints the message: "Model loaded. Resuming from epoch {checkpoint['num\_epochs']}." to indicate which epoch it will resume training from.
  + Finally, the function returns the epoch number from which training can resume using return checkpoint['num\_epochs'].

If the Saved Model Does Not Exist:

* If the file 'attention\_model\_state.pth' does not exist:
  + It prints "No saved model found. Training from scratch.", indicating that training will start fresh.
  + Returns 1, which means training should start from epoch 1.

Usage of the Function:

* start\_epoch = load\_model(model): This line calls the load\_model function and stores the returned epoch number in start\_epoch. Depending on whether a saved model was found or not, training will either resume from a specific epoch or start from the first epoch.

**18. Training Hyperparameters:**

num\_epochs = 25

print\_every = 100

* **num\_epochs = 25**: The model will train for 25 full passes over the dataset.
* **print\_every = 100**: Every 100 batches, the training loss will be printed, and a caption will be generated for one image to monitor the model’s progress.

**19. Training Loop:**

The loop begins by iterating over epochs and batches in the dataset:

for epoch in range(1, num\_epochs+1):

for idx, (image, captions) in enumerate(iter(data\_loader)):

image, captions = image.to(device), captions.to(device)

* **for epoch in range(1, num\_epochs+1)**: Loop through each epoch (from 1 to 25).
* **for idx, (image, captions) in enumerate(iter(data\_loader))**: Iterate through each batch of images and captions from the data\_loader.

**20. Zero the Gradients:**

# Zero the gradients.

optimizer.zero\_grad()

* **optimizer.zero\_grad()**: Clears old gradients from the last step. This is necessary because gradients are accumulated in PyTorch by default.

**21. Forward Pass:**

# Feed forward

outputs, attentions = model(image, captions)

* **outputs, attentions = model(image, captions)**:
  + The input **images** and **captions** are passed through the model. The model consists of an encoder (CNN) and decoder (LSTM with attention).
  + **outputs**: These are the predicted word probabilities (i.e., the model's predictions for the next word in the sequence).
  + **attentions**: These are the attention weights, which show where the model is focusing in the image while generating each word in the caption.

**22. Compute Loss:**

# Calculate the batch loss.

targets = captions[:, 1:]

loss = criterion(outputs.view(-1, vocab\_size), targets.reshape(-1))

* **targets = captions[:, 1:]**: The targets are the ground-truth captions, starting from the second word onward (i.e., the caption without the <SOS> start token). This ensures the model predicts the next word.
* **loss = criterion(outputs.view(-1, vocab\_size), targets.reshape(-1))**:
  + The **cross-entropy loss** is calculated between the model’s predictions (outputs) and the true captions (targets).
  + **outputs.view(-1, vocab\_size)**: The predictions are reshaped so that the output of all time steps in the sequence is flattened for easier comparison with the targets.
  + **targets.reshape(-1)**: The targets are also reshaped to match the dimensions of the predictions.

**23. Backward Pass and Optimization:**

# Backward pass.

loss.backward()

# Update the parameters in the optimizer.

optimizer.step()

* **loss.backward()**: Performs the backward pass to compute gradients based on the loss.
* **optimizer.step()**: Updates the model's parameters using the computed gradients. This is where the actual learning happens.

**24. Print Loss and Generate Captions Periodically:**

if (idx+1) % print\_every == 0:

print("Epoch: {} loss: {:.5f}".format(epoch, loss.item()))

# Generate the caption

model.eval()

with torch.no\_grad():

dataiter = iter(data\_loader)

img, \_ = next(dataiter)

features = model.encoder(img[0:1].to(device))

caps, alphas = model.decoder.generate\_caption(features, vocab=dataset.vocab)

caption = ' '.join(caps)

show\_image(img[0], title=caption)

model.train()

* **Print loss**:
  + Every print\_every batches (i.e., every 100 batches), the current epoch and loss are printed to monitor training progress.
* **Generate caption**:
  + The model switches to **evaluation mode** (model.eval()) to generate captions for an image. In evaluation mode, certain layers like dropout or batch normalization behave differently.
  + **torch.no\_grad()**: Ensures no gradients are calculated (this reduces memory usage during inference).
  + **dataiter = iter(data\_loader)**: A new iterator is created from the data\_loader, and one batch of images is fetched (img, \_ = next(dataiter)).
  + The image features are extracted by the **encoder** (model.encoder(img[0:1].to(device))).
  + The **decoder** generates a caption from the image features (caps, alphas = model.decoder.generate\_caption(features, vocab=dataset.vocab)).
  + The generated caption is displayed along with the image using the **show\_image** function (show\_image(img[0], title=caption)).
* The model is switched back to **training mode** after generating captions (model.train()).

**25. Save the Model:**

# Save the latest model

save\_model(model, epoch)

* After each epoch, the model is saved using the **save\_model** function, which stores the model's state (weights, hyperparameters, etc.) for later use. The model is saved to a file named attention\_model\_state.pth.

**26. Generating the Caption (get\_caps\_from function):**

def get\_caps\_from(features\_tensors):

# generate the caption

model.eval()

with torch.no\_grad():

features = model.encoder(features\_tensors.to(device))

caps, alphas = model.decoder.generate\_caption(features, vocab=dataset.vocab)

caption = ' '.join(caps)

show\_image(features\_tensors[0], title=caption)

return caps, alphas

* **Inputs:**
  + features\_tensors: A tensor representing the input image(s) (batch of features) that will be fed to the model.
* **Process:**
  + model.eval(): Switches the model into evaluation mode (important when using dropout or batch normalization layers).
  + with torch.no\_grad(): Disables gradient calculation since it's not needed during inference, which saves memory and speeds up computations.
  + model.encoder(features\_tensors.to(device)): Passes the image features to the model's encoder (likely a convolutional neural network) and moves the tensor to the proper device (e.g., CPU or GPU). This generates encoded image features.
  + model.decoder.generate\_caption(features, vocab=dataset.vocab): The decoder part of the model generates the caption, given the encoded image features. It returns two values:
    - caps: The generated caption (list of words).
    - alphas: The attention weights corresponding to the caption generation process, where attention focuses on certain parts of the image for each word.
  + caption = ' '.join(caps): Joins the list of words (caps) into a single string for readability.
  + show\_image(features\_tensors[0], title=caption): Displays the image along with the generated caption.
* **Outputs:**
  + The function returns caps (caption tokens) and alphas (attention maps).

**27. Plotting the Attention (plot\_attention function):**

def plot\_attention(img, result, attention\_plot):

# Untransform the image (reverse normalization)

img[0] = img[0] \* 0.229

img[1] = img[1] \* 0.224

img[2] = img[2] \* 0.225

img[0] += 0.485

img[1] += 0.456

img[2] += 0.406

img = img.numpy().transpose((1, 2, 0))

temp\_image = img

fig = plt.figure(figsize=(15, 15))

len\_result = len(result)

for l in range(len\_result):

temp\_att = attention\_plot[l].reshape(7, 7)

ax = fig.add\_subplot(len\_result//2, len\_result//2, l+1)

ax.set\_title(result[l])

img = ax.imshow(temp\_image)

ax.imshow(temp\_att, cmap='gray', alpha=0.7, extent=img.get\_extent())

plt.tight\_layout()

plt.show()

* **Inputs:**
  + img: The input image tensor (possibly normalized).
  + result: The generated caption (list of words).
  + attention\_plot: The attention weights or maps for each word in the caption.
* **Process:**
  + **Reverse Normalization:** The image tensor is initially normalized (a common preprocessing step for neural networks). The first part of the function multiplies the image tensor's channels (R, G, B) by the standard deviation used in normalization (0.229, 0.224, 0.225) and then adds the mean (0.485, 0.456, 0.406) to reverse the normalization process, converting the tensor back to its original pixel values.
  + img = img.numpy().transpose((1, 2, 0)): Converts the image tensor into a NumPy array and changes the channel order from (C, H, W) to (H, W, C) to display it properly using matplotlib.
  + **Plot Setup:** Creates a figure for plotting with fig = plt.figure(figsize=(15, 15)).
  + **Plotting Attention:** Loops through each word in the caption (len\_result = len(result)) and plots the attention map for each word:
    - temp\_att = attention\_plot[l].reshape(7, 7): Reshapes the attention map for each word into a 7x7 grid.
    - ax.imshow(temp\_image): Displays the original image.
    - ax.imshow(temp\_att, cmap='gray', alpha=0.7, extent=img.get\_extent()): Overlays the attention map on top of the image, with a grayscale colormap and some transparency (alpha=0.7) to visualize where the model is focusing for that word.
* **Outputs:**
  + Displays the image with attention maps for each word in the generated caption.

**28. Visualizing Attention and Generating the Caption:**

dataiter = iter(data\_loader)

images, \_ = next(dataiter)

img = images[0].detach().clone()

img1 = images[0].detach().clone()

caps, alphas = get\_caps\_from(img.unsqueeze(0))

plot\_attention(img1, caps, alphas)

* **Process:**
  + dataiter = iter(data\_loader): Retrieves an iterator from the data loader, which contains batches of images.
  + images, \_ = next(dataiter): Gets the next batch of images.
  + img = images[0].detach().clone(): Detaches the first image from the computation graph and clones it. This ensures the original image tensor won't be altered during the process.
  + img1 = images[0].detach().clone(): Clones the image again for displaying later.
  + caps, alphas = get\_caps\_from(img.unsqueeze(0)): Generates the caption and attention maps for the first image by passing it to the get\_caps\_from function. The unsqueeze(0) adds a batch dimension since the model expects batches.
  + plot\_attention(img1, caps, alphas): Plots the attention visualization for the image using the generated caption (caps) and attention maps (alphas).